

02

BACKGROUND AND STATE OF THE ART

■ 2.1 WebAssembly

The W3C publicly announced the WebAssembly language in 2017 as the four scripting language supported in all major web browser vendors. Wasm is a binary instruction format for a stack-based virtual machine and was officially consolidated by the work of Haas et al. [?] in 2017. Wasm is designed to be fast, portable, self-contained and secure, and it promises to outperform JavaScript execution [?].

Since 2017, the adoption of Wasm keeps growing. For example; Adobe, announced a full online version of Photoshop¹ written in WebAssembly; game companies moved their development from JavaScript to Wasm like is the case of a full Minecraft version². Moreover, WebAssembly has been evolving outside web browsers since its first announcement. Some works demonstrated that using WebAssembly as an intermediate layer is better in terms of startup and memory usage than containerization and virtualization [? ?]. Consequently, in 2019, the Bytecode Alliance [?] proposed WebAssembly System Interface (WASI) [?]. WASI pioneered the execution of Wasm with a POSIX system interface protocol, making possible to execute Wasm directly in the operating system. Therefore, it standardizes the adoption of Wasm in heterogeneous platforms [?], making it suitable standalone and backend execution scenarios [? ?].

■ 2.1.2 WebAssembly's generation

WebAssembly programs are pre-compiled from source languages like C/C++, Rust, or Go, which means that it can benefit from the optimizations of the source language compiler. The resulting Wasm program is like a traditional shared library, containing instruction codes, symbols, and exported functions. A host environment is in charge of complementing the Wasm program, such as providing external functions required for execution within the host engine. For instance, functions for interacting with an HTML page's DOM are imported into the Wasm binary when invoked from JavaScript code.

¹<https://twitter.com/Adobe/status/1453034805004685313?s=20&t=Zf1N7-WmzecA0K4V8R69lw>

²<https://satoshinm.github.io/NetCraft/>

TODO Expand on how we generate it

In Listing 2.1 and Listing 2.2, we illustrate a C program and its corresponding Wasm binary. The C function includes heap allocation, external function usage, and a function definition featuring a loop, conditional branching, function calls, and memory accesses. The Wasm code in Listing 2.2 displays the textual format of the generated Wasm (Wat).

```
// Some raw data
const int A[250];

// Imported function
int ftoi(float a);

int main() {
    for(int i = 0; i < 250; i++) {
        if (A[i] > 100)
            return A[i] + ftoi(12.54);
    }

    return A[0];
}
```

Listing 2.1: Example C program which includes heap allocation, external function usage, and a function definition featuring a loop, conditional branching, function calls, and memory accesses.

```
; WebAssembly magic bytes(\0asm) and version (1.0) ;
(module
; Type section: 01 ...
  (type (;0;) (func (param f32) (result i32)))
  (type (;1;) (func))
  (type (;2;) (func (result i32)))
; Import section: 02 ...
  (import "env" "ftoi" (func $ftoi (type 0)))
; Code section: 03 ...
  (func $main (type 2) (result i32)
    (local i32 i32)
    i32.const -1000
    local.set 0
    block ;label = @1;
    loop ;label = @2;
      i32.const 0
      local.get 0
      i32.add
      i32.load
      local.tee 1
      i32.const 101
      i32.ge_s
      br_if 1 ;@1;
      local.get 0
      i32.const 4
      i32.add
      local.tee 0
      br_if 0 ;@2;
    end
    i32.const 0
    return
  end
  f32.const 0x1.9147aep+3
  call $ftoi
  local.get 1
  i32.add)
; Memory section: 05 ...
  (memory (;0;) 1)
; Global section: 06 ...
  (global (;4;) i32 (i32.const 1000))
; Export section: 07 ...
  (export "memory" (memory 0))
  (export "A" (global 2))
; Data section: 0d ...
  (data $data (0) "\00\00\00\00...")
)
```

Listing 2.2: Wasm code for Listing 2.1. The example Wasm code illustrates the translation from C to Wasm in which several high-level language features are translated into multiple Wasm instructions.

■ 2.1.3 WebAssembly's binary format

The Wasm binary format is close to machine code and already optimized. Thus, its consuming process typically involves a straightforward one-to-one mapping. For example, a compiler might accelerate the compilation process by parallelizing the parsing process. The main reason behind this claim is that a Wasm binary is organized into a contiguous collection of sections. In Figure 2.1 we show the binary format of a Wasm section. A Wasm section starts with a 1-byte section ID, followed by an 8-byte section size, and concludes with the section content, which precisely matches the size indicated earlier.

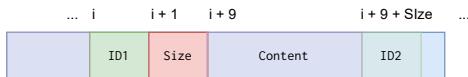


Figure 2.1: Memory byte representation of a WebAssembly binary section, starting with a 1-byte section ID, followed by an 8-byte section size, and finally the section content.

A Wasm binary contains sections of 12 types, each with a specific semantic role and placement within the module. Each section is optional, where an omitted section is considered empty. In the following text, we summarize each one of the 12 types of Wasm sections, providing their name, ID, and purpose. In addition, some sections are annotated as comments in the Wasm code in Listing 2.2.

TODO Check if better with examples in each section

Custom Section (00) : Comprises two parts: the section name and arbitrary content. Primarily used for storing metadata, such as the compiler used to generate the binary. This type of section has no order constraints with other sections and is optional. Compilers usually skip this section when consuming a WebAssembly binary.

Type Section (01) : Contains the function signatures for functions declared or defined within the binary. It must occur only once in a binary. It can be empty.

Import Section (02) : Lists elements imported from the host, including functions, memories, globals, and tables. It must occur only once in a binary. It can be empty.

Function Section (03) : Details functions defined within the binary. It essentially maps Type section entries to Code section entries. It must occur only once in a binary. It can be empty.

Table Section (04) : Groups functions with identical signatures to control indirect calls. It must occur only once in a binary. It can be empty.

Memory Section (05) : Specifies the number and initial size of unmanaged linear memories. It must occur only once in a binary. It can be empty.

Global Section (06) : Defines global variables as managed memory for use and sharing between functions in the WebAssembly binary. It must occur only once in a binary. It can be empty.

Export Section (07) : Declares elements like functions, globals, memories, and tables for host engine access. The entry point of the WebAssembly binary is typically declared here. It must occur only once in a binary. It can be empty.

Start Section (08) : Designates a function to be called upon binary readiness, initializing the WebAssembly program state before executing any exported functions. It must occur only once in a binary. It can be empty.

Element Section (09) : Contains elements to initialize the binary tables. It must occur only once in a binary. It can be empty.

Code Section (10) : Contains the body of functions defined in the Function section. Each entry consists of local variables used and a list of instructions. It must occur only once in a binary. It can be empty.

Data Section (11) : Holds data for initializing unmanaged linear memory. Each entry specifies the offset and data to be placed in memory. It must occur only once in a binary. It can be empty.

Data Count Section (12) : Primarily used for validating the Data Section. If the segment count in the Data Section mismatches the Data Count, the binary is considered malformed. It must occur only once in a binary. It can be empty.

- 2.1.4 WebAssembly's Runtime structure

TODO Reorder this text according with the wrules

The WebAssembly runtime structure is described in the WebAssembly specification by enunciating 10 key components: the Store, Stack, Locals, Module Instances, Function Instances, Table Instances, Memory Instances, Global Instances, Export Instances, and Import Instances. These components are particularly significant in maintaining the state of a WebAssembly program during its execution. In the following text, we provide a brief description of each runtime component. Notice that, the runtime structure is an abstraction that serves to validate Wasm consumers.

Store : The WebAssembly store represents the global state and is a collection of instances of functions, tables, memories, and globals. Each of these instances is uniquely identified by an address, which is usually represented as an i32 integer.

Module Instances : A module instance is a runtime representation of a loaded and initialized WebAssembly module. It contains the runtime representation of all the definitions within a module, including functions, tables, memories, and globals, as well as the module’s exports and imports.

Table instances : A table instance is a vector of function elements. WebAssembly tables are used to support indirect function calls. For example, it allows modeling dynamic calls of functions (through pointers) from languages such as C/C++, for which the Wasm’s compiler is in charge of populating the static table of functions.

Export Instances : Export instances represent the functions, tables, elements, globals or memories that are exported by a module.

Import Instances : Import instances represent the functions, tables, elements, globals or memories that are imported into a module from the host.

The Stack holds typed values and control frames, with control frames handling block instructions, loops, and function calls. Values inside the stack can be of the only static types allowed in Wasm 1.0, `i32` for 32 bits signed integer, `i64` for 64 bits signed integer, `f32` for 32 bits float and `f64` for 64 bits float. Therefore, abstract types, such as classes, objects, and arrays, are not natively supported. Instead, during compilation, such types are transformed into primitive types and stored in the linear memory.

Memory Instances represent the unmanaged linear memory of a WebAssembly program, consisting of a contiguous array of bytes. Memory instances are accessed with `i32` pointers (integer of 32 bits). Memory instances are usually bound in browser engines to 4Gb of size, and it is only shareable between the process that instantiates the WebAssembly module and the binary itself.

Global Instances : A global instance is a global variable with a value and a mutability flag, indicating whether the global can be modified or is immutable. Global variables are part of the managed data, i.e., their allocation and memory placement are managed by the host engine. Global variables are only accessible by their declaration index, and it is not possible to dynamically address them.

Locals : Locals are mutable variables that are local to a specific function invocation. As globals, locals are part of the managed data.

Note 1. Along with this dissertation, as the work of Lehmann et al. [?], we refer to managed and unmanaged data to differentiate between the data that is managed by the host engine and the data that is managed by the WebAssembly program respectively.

Function Instances : A function instance groups locals and a function body.

Locals are typed variables that are local to a specific function invocation. The function body is a sequence of instructions that are executed when the function is called. Each instruction either reads from the stack, writes to the stack, or modifies the control flow of the function. Recalling the example Wasm binary previously showed, the local variable declarations and typed instructions that are evaluated using the stack can be appreciated between Line 7 and Line 32 in Listing 2.2. Each instruction reads its operands from the stack and pushes back the result. In the case of Listing 2.2, the result value of the main function is the calculation of the last instruction, `i32.add` at result. As the listing shows, instructions are annotated with a numeric type.

■ 2.1.5 WebAssembly's control flow

In WebAssembly, a defined function instructions are organized into blocks, with the function's starting point serving as the root block. Unlike traditional assembly code, control flow structures in Wasm jump between block boundaries rather than arbitrary positions within the code. Each block might specify the required stack state before execution and the resulting stack state after its instructions have run. This stack state is used to validate the binary during compilation and to ensure that the stack is in a valid state before executing the block's instructions. Blocks in Wasm are explicit, indicating, where they start and end. By design, each block cannot reference or execute code from outer blocks.

Control flow within a function is managed through three types of break instructions: unconditional break, conditional break, and table break. Importantly, each break instruction is limited to jumping to one of its enclosing blocks. Unlike standard blocks, where breaks jump to the end of the block, breaks within a loop block jump to the block's beginning, effectively restarting the loop. To illustrate this, Listing 2.3 provides an example comparing a standard block and a loop block in a Wasm function.

```

block
  block
    br 1 ; Jump instructions
           ; are annotated with the
           ; depth of the block they
           ; jump to;
    end
  ...
end
...                                     loop
  ...
  br 0 ; first-order break;
  ...
end ; end instructions break
      ; the block and jump to next
      ; instruction;
  ...

```

Listing 2.3: Example of breaking a block and a loop in WebAssembly.

Each break instruction includes the depth of the enclosing block as an operand. This depth is used to identify the target block for the break instruction. For example, in the left-most part of the previously discussed listing, a break instruction with a depth of 1 would jump past two enclosing blocks. For the

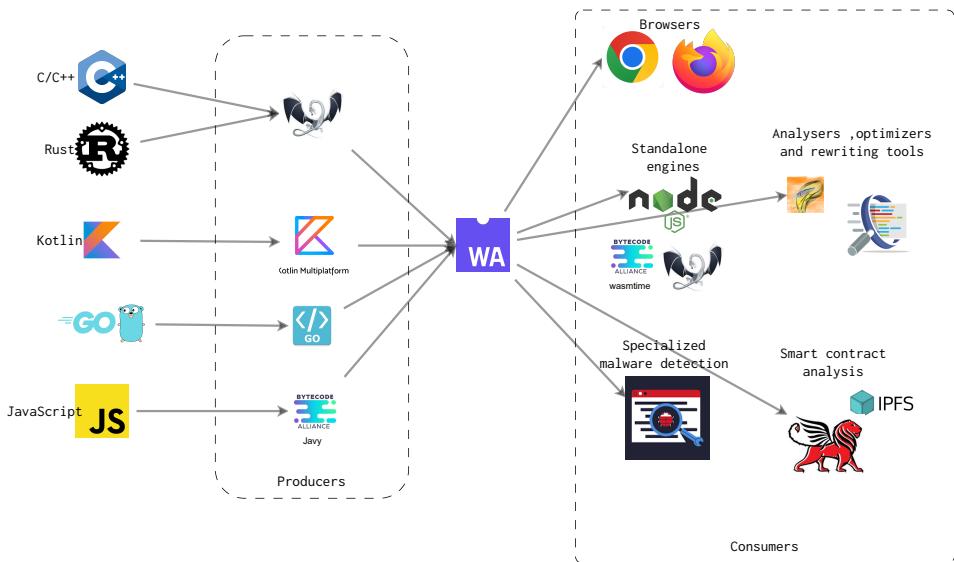


Figure 2.2: WebAssembly’s ecosystem landscape separated into producers and consumers. For the sake of simplicity we do not include each tool mentioned in this chapter.

purposes of this dissertation, we introduce a specific term to describe a particular kind of break within loops:

Definition 1. *Break instructions within loops that effectively jump to the loop’s beginning are termed first-order breaks.*

■ 2.1.6 WebAssembly’s Ecosystem

WebAssembly programs are designed for execution in host environments such as web browsers. Though the execution of a WebAssembly program might be considered its final lifecycle stage, the WebAssembly ecosystem is far from simplistic. It comprises multiple stakeholders and a rich array of tools that cater to various needs [?]. In Figure 2.2 we simplify the ecosystem landscape by separating it into producers, consumers and major stakeholders categories. In the subsequent text, we describe the WebAssembly ecosystem by separating it into stakeholders categories.

Producers, such as compilers, transform source code into WebAssembly binaries. For example, LLVM has offered WebAssembly as a backend option since its 7.1.0 release³, supporting a diverse set of frontend languages like C/C++,

³<https://github.com/llvm/llvm-project/releases/tag/llvmorg-7.1.0>

Rust, Go, and AssemblyScript⁴. In parallel developments, the KMM framework⁵ has incorporated WebAssembly as a compilation target, and the Javy approach⁶ focuses on encapsulating JavaScript code within isolated WebAssembly binaries. This is achieved by porting both the engine and the source code into a secure WebAssembly environment. Blazor also enables the compilation of C code into WebAssembly binaries for browser execution⁷. Regardless of the source language or framework, the resulting WebAssembly binary functions similar to a traditional shared library, replete with code instructions, symbols, and exported functions.

Consumers encompass tools that undertake the tasks of validating, analyzing, optimizing, transpiling to machine code, and executing WebAssembly binaries, e.g. browser clients. In the text that follows, we dissect them into specific categories and their respective domains of application. Notice that, while some tools are designed for a specific domain, others are more general-purpose and might encapsulate more than one task in the WebAssembly ecosystem. For example, this is the case of browsers and standalone engines, which in one way or the other perform each one of the previous tasks.

Browser engines like V8[?], SpiderMonkey[?], and Chakra[?] are at the forefront of executing WebAssembly binaries in browser clients. These engines leverage Just-In-Time (JIT) compilers to convert WebAssembly into machine code. This translation is typically a straightforward one-to-one mapping, given that WebAssembly is already an optimized format closely aligned with machine code, as previously discussed in Subsection 2.1.3. For example, V8 employs quick, rudimentary optimizations such as constant folding and dead code removal⁸.

Standalone engines WebAssembly’s applicability has transcended browser environments, largely due to the WebAssembly System Interface (WASI)[?]. WASI aims to standardize the interaction between host environments and WebAssembly modules, thereby facilitating portability of both bytecode and binaries across diverse platforms. It outlines a POSIX-like Application Binary Interface (ABI), which includes functionalities for filesystem access, network sockets, clocks, and random number generation, among others. Standalone engines such as WASM3[?], wasmtime[?], WAVM[?] and Sledge [?] have emerged to support WebAssembly and WASI. Similarly, Singh and colleagues [?] proposed a virtual machine for Wasm in Arduino based devices.

TODO Disect them into, JIT compilers and interpreters, ending with SWAM.

TODO Add the verifiable standalone engine here

⁴A subset of the TypeScript language

⁵<https://kotlinlang.org/docs/wasm-overview.html>

⁶<https://github.com/bytocodealliance/javy>

⁷<https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor>

⁸This analysis was corroborated through discussions with the V8 development team and through empirical studies in one of our contributions[?]

TODO Locate WasmA [?]

Static and dynamic analysis, optimization and validation: Tools such as Wassail[?], Wasmati[?], and Wasp[?] utilize a range of strategies, including information flow control, code property graphs, and concolic execution, to identify vulnerabilities taking Wasm programs as input. Dynamic analysis counterparts like TaintAssembly[?], Wasabi[?], and Fuzzm[?] serve analogous functions.

TODO Binanryen **TODO** <https://soft.vub.ac.be/Publications/2023/vub-tr-soft-23-11.pdf>, https://dl.acm.org/doi/abs/10.1145/351003.3510070?casa_token=hAlPiu9_oioAAAAA:DskwqdqM8c5-4e4W-8uU2CiENeyyOXWoi6BHytTiZUgyxojsYWXBsaS7ERFeqGNKfaYIKIN8pnQ1 **TODO** Waf

TODO Add veriwasm **TODO** Wasmfuzzer

Specialized Malware Detection In niche areas like cryptomalware detection, tools like MineSweeper[?], MinerRay[?], and MINOS[?] utilize static analysis techniques. Conversely, dynamic analysis is the forte of tools like SEISMIC[?], RAPID[?], and OUTGuard[?].

Smart Contract Analysis In the field of smart contracts, static analysis tools like EvalHunter[?], WANA[?], and EOSAFE[?] are employed to unearth vulnerabilities in WebAssembly-based contracts. Dynamic analysis tools in this sphere include EOSFuzzer[?] and wasai[?].

Obfuscation, and binary rewriting tools [?][?] **TODO** WASMixer, Wobfuscator

While many existing tools purport to offer self-correctness evaluations and exhaustive test suites, the WebAssembly ecosystem is still in its early stages. To emphasize this, a 2021 study by Hilbig et al.[?] found a mere 8,000 unique WebAssembly binaries globally. This pales in comparison to mature ecosystems like JavaScript and Python, which offer 1.5 million and 1.7 million packages on npm⁹ and PyPI¹⁰, respectively. This limited pool of WebAssembly binaries presents a unique challenge for machine learning-based analysis tools, which require large datasets for effective training. This issue is explicitly addressed in one of the contributions of this dissertation[?]. Additionally, the scarcity of WebAssembly programs exacerbates the problem of software monoculture, increasing the likelihood of consuming a compromised WebAssembly program[?]. Besides, the current size of the WebAssembly ecosystem offers a small testing environment for evaluating the capabilities of consumer tools. This dissertation aims to address these issues by introducing a comprehensive suite of tools. These tools are crafted to not only bolster the security of WebAssembly programs through Software Diversification but also to intensify the testing rigor for both producers and consumers in the ecosystem.

⁹<https://www.npmjs.com/>

¹⁰<https://pypi.org/>

- 2.1.7 WebAssembly’s Security

TODO Check the attack primitives defined by Lehmann. **TODO** Extend with more cases !

As we described, Wasm is deterministic and well-typed, follows a structured control flow and explicitly separates its linear memory model, global variables and the execution stack. This design is robust [?] and makes it easy for compilers and engines to sandbox the execution of Wasm binaries. Following the specification of Wasm for typing, memory, virtual stack and function calling, host environments should provide protection against data corruption, code injection, and return-oriented programming (ROP).

However, implementations in both browsers and standalone runtimes [?] are vulnerable. Genkin et al. demonstrated that Wasm could be used to exfiltrate data using cache timing-side channels [?]. Moreover, binaries itself can be vulnerable. The work of Lehmann et al. [?] proved that C/C++ source code vulnerabilities can propagate to Wasm such as overwriting constant data or manipulating the heap by overflowing the stack. Even though these vulnerabilities need a specific standard library implementation to be exploited, they make a call for better defenses for Wasm. Recently, Stiévenart and colleagues demonstrate that C/C++ source code vulnerabilities can be ported to Wasm [?]. Several proposals for extending Wasm in the current roadmap could address some existing vulnerabilities. For example, having multiple memories¹¹ could incorporate more than one memory, stack and global spaces, shrinking the attack surface. However, the implementation, adoption and settlement of the proposals are far from being a reality in all browser vendors¹².

- 2.2 Software diversification

- 2.2.2 Generating Software Diversification
- 2.2.3 Variants generation
- 2.2.4 Variants equivalence

TODO Automatic, SMT based **TODO** Take a look to Jackson thesis, we have a similar problem he faced with the superoptimization of NaCL **TODO** By design **TODO** Introduce the notion of rewriting rule by Sasnaukas. https://link.springer.com/chapter/10.1007/978-3-319-68063-7_13

¹¹<https://github.com/WebAssembly/multi-memory/blob/main/proposals/multi-memory/Overview.md>

¹²<https://webassembly.org/roadmap/>

- 2.3 Exploiting Software Diversification
 - 2.3.2 Defensive Diversification
 - 2.3.3 Offensive Diversification